Introduction to Big Data

Lesson 1.3 High-performance computing without distribution

Arthur Katossky & Rémi Pépin

Tuesday, March 29, 2023

High-performance computing without distribution

Shift focus to relative precision

Shift focus away from absolute precision to **relative** or **marginal precision** :

- per additional observation
- per additional second of computation
- per additional dollar used
- per additional Watt-hour spent

Profile code

Try to identify **bottlenecks** in **memory use**, **computing time**, network exchange, etc. and how they evolve with the problem size.

- python :
  - computing time : `cProfile`
  - memory : `memory_profiler`
- java : `JProfiler`
- R : `profvis`

Profile code

Demo

Analyse code

After having identified a bottleneck, especially in computing time

- algorithmic analysis can confirm the profiling ($O(n^2)$, $O(n \ln(n))$, $O(n!)$, etc.)
- known problems have known approximations: matrix inversion, matrix cross product, spectral decomposition, etc.

Use approximate solutions

Algorithmic approximation errors are often orders of magnitude smaller than:

- rounding floating-point errors
- statistical uncertainty

Well know approximation algorithm :

- stochastic gradient decent (ML)
- approximation kNN
- greedy algorithm
- Matrix spectral approximation

Down-size

- select (columns) and filter (rows)
- sample

In development phases, work on small data. In production, only load what you really need.

Shift from memory to disk

- you usually have much more disk space
- need to use some special library or design your own algorithms
- usually increases the computation time (hard drives are slower than RAM)

Store or process data in chunks

Traditional **batch processing** consists in processing a whole data set simultaneously.

When this causes issues, shift to processing data in **chunks** (per smaller groups of observations) or in **streams** (one at a time).

- comes at no cost with **associative symmetrical operations** (like summation) but requires substantial rewriting for other algorithms
- **online algorithms** (that can be updated one element at a time) are typical examples

There is an ambiguity with the word **batch** that sometimes refer to the whole data set, sometimes to a subset.

Take advantage of sparcity

A **sparse** data set is a data set with low information content.

In storage, you can have default values for each column and **only store the positions and values of non-defaults**. Or you can use integers to represent a fixed number of character strings, like factors do in R.

```
char <- sample(
  size = 2000, replace = T,
  c("A first quite long string", "Each string takes serval octets to store")
)

object.size(char)
```

```
## 16224 bytes
```

```
object.size(factor(char))
```

```
## 8624 bytes
```

Take advantage of sparcity

In computation, it is possible to perform **sparse matrix operations** very effectively. Package `scipy.sparse` in python. Multiple formats, each has pros/cons.

```r
library(Matrix) # sparse package in R

n <- 500
a <- runif(n**2)
a <- ifelse(runif(n**2)<0.9, 0, a)
A <- matrix(nrow=n, ncol=n, a)
B <- Matrix(A, sparse = T)
```

```r
object.size(A)
```

```
## 2000216 bytes
```

```r
object.size(B)
```

```
## 304960 bytes
```

```r
microbenchmark::microbenchmark(A %*% t(A), B %*% t(B) )
```

```
## Unit: milliseconds
##        expr       min        lq      mean    median        uq       max neval
##  A %*% t(A) 41.214266 41.452784 41.91871 41.644458 41.849725 45.662192   100
##  B %*% t(B)  5.229878  5.372394  5.80262  5.523828  5.743383  8.774369   100
```

Go low-level

Python, R or spreadsheets are **high-level programs**

- tailored for human understanding (**expressive power**)
- abstract away the actual processing by the machine (**transparency** [1])

However, you also lose control on the fine details of execution

[1] In computer science, transparency means that you *don't see* the machine, not that you *see* it *perfectly*. A "transparent" process hides technical implementations away.

Go low-level

High-level programs:

- deliver good human experience (column names, error messages, warnings)
- perform a lot a checks and formatting operations
- have lower-level sub-functions that only accept very standardized inputs

In R, the `lm()` function is actually only tests and formatting, the actual computation happening inside a call to `lm.fit()`, which itself calls the C function `Cdqrls`.

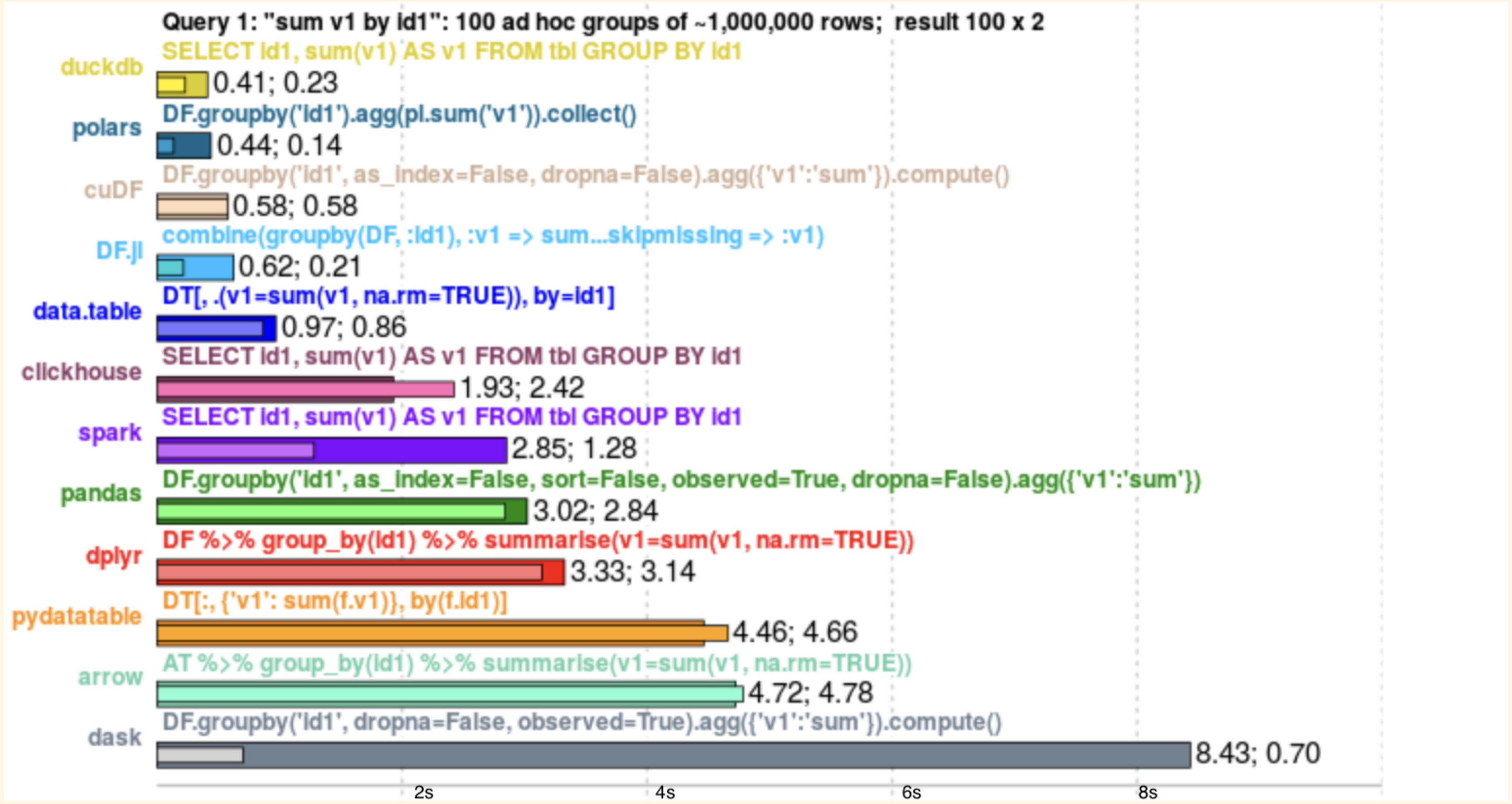
Skipping formatting and tests can increase the speed! Especially if you call such functions multiple times.

Go low-level

Other languages, most notably **C**, **C++** and **Fortran**, are much closer to machine code and allow better control over the flow of information between the disk, the memory and the processors.

R or Python allow you to write functions directly in these languages, with often vast speed improvements.

Go low-level



Source: https://h2oai.github.io/db-benchmark/

Go low-level

Command line code, also known as **bash** or **shell**, is a way to take advantage of very effective low-level functions.

You can execute command line code from R with `system(...)` or with Python through `os.system(...)` or `subprocess.run(...)`.

A usually impressively fast data but hard-to-learn processing command line is the infamous **`awk`**.

> Recently I got put in change of wrangling 25+ TB of raw genotyping data for my lab. When I started, using spark took 8 min & cost $20 to query a SNP. After using AWK + #rstats to process, it now takes less than a 10th of a second and costs $0.00001. My personal #BigData win. pic.twitter.com/ANOXVGrmkk
>
> — Nick Strayer (@NicholasStrayer) May 30, 2019

Go low-level : Compilation

The lowest-level code is **machine code**, specific to each machine and high-level programs have to be **translated** into machine code :

1. either **interpreting** the source code : instructions from source code are converted **one by one** into machine code
2. or **compiling** the source code : instruction from source code are converted **as a whole** into machine code, while performing many optimisation procedures under the hood

The real world is not black and white. For instance most interpreters do perform optimisations on the fly, even though not as thorough as compilers.

Go low-level : compilation

Cython compiles python code to C.

1. Static type your code
2. Compile to C
3. Import your C code as an external library

Not magical, but easy-to-reach improvement.

Go low-level : compilation

A common intermediate pattern is **compilation to bytecode**, a platform agnostic language that gets interpreted at run time, or even further compiled to machine code.

Compiled code often faster than non-compiled one but **compiling** also **takes time**. It is usually worth it only when you reuse the same peace of code several times.

Go low-level : building chips

Building a new electronic component is the lowest-level one can go.

Move computing at the data source

- with remote data, filter, select and sample at a distance (no `SELECT * FROM table`)
- if possible execute code distantly and return results (prepared statement)
- results usually have smaller size than data

For instance, most SQL servers have (limited) statistical abilities for mean, sum, percentiles, variance, standard error, etc. More recently, many have moved to allow execution of arbitrary scripts in R and Python.

On the other hand, `dbplyr` allows you to write plain R code (tidyverse-style) and then optimises the SQL request in order to move as much treatment as possible at the source.

Pipeline i/o operations

- moving data around takes time, even inside a computer
- reading (load) and writing (save) operations may be bottlenecks
- if possible, perform readings / writings in parallel to earlier / later tasks, aka. **pipelining**

A common mistake is printing messages inside a fast loop. The printing actually slows down the whole process. Printing once every k steps may be preferable.

Use cache

- avoid downloading / reading data twice
- avoid performing the same computation twice
- save results that you want to reuse later

In the cloud, you actually pay for most data transfers!

Use cache

```r
library(memoise)

fib <- function(x) {
  if (x == 0) return(0)
  if (x == 1) return(1)
  Recall(x - 1) + Recall(x - 2) # recursive
}

fib <- memoise(fib)

system.time(fib(30)) # first call isn't cached
```

```
##    user  system elapsed 
##   0.851   0.010   0.863
```

```r
system.time(fib(30)) # second called is
```

```
##    user  system elapsed 
##   0.007   0.000   0.007
```